References and Borrowing

Ownership

Rust has it's own way of managing memories that are allocated on the heap. Unlike C, where the burden of allocating and freeing the memory that is allocated on the heap falls on the shoulder of the programmer, Rust manages the memory for you as long as you follow it's ownership conventions.

This is how Rust deal with dynamically allocated memory on the heap. When you allocate data on the heap, that piece of data will be associated with a variable name. When the scope of that variable ends, and naturally (most of the time) the data on the heap associated with that variable will need to be freed, and Rust does that for you automatically:

```
{
let s = String::from("Hello"); // s is valid from this point forward
}// Scope of s is over, s is no longer valid, and the memory is returned to the allocator
```

Whenever a variable goes out of scope, if it has its memory allocated on the heap, it will be called a special function called `drop` when the variable goes out of scope. **It will free the memory that is allocated for that variable back to the allocator, Rust will call it for you automatically at the closing curly bracket (the end of a scope).**

Data move

Let's look at some form of alias and copying in Rust:

```
let x = 5;
let y = x;
```

This will do what you expect, the value of `x` is copied over to `y`. If you change `y` it will not affect the value of `x`.

Now let's look at data allocated on the heap.

```
let s1 = String::from("Hello");
let s2 = s1;
```

If we do this, we assume that `s2` is an alias which points to the same string that's allocated on the heap. That will be true for language that C or Python, but in Rust it is different. When you make an alias to another data allocated on the heap, that pointer will be moved from `s1` to `s2`, so `s1` will no longer be valid after line number 2! This is due to the design of Rust because like we have mentioned earlier, when a variable goes out of scope a special `drop` function is called to free up the memory allocated for the data, if there is two variables that points to that allocated data, then there is going to be a double free error!

To resolve this, Rust does variable move, so when you do `s2 = s1`, `s1` will no longer be valid afterward, so Rust only needs to worry about freeing up `s2` and doesn't have to worry about freeing `s1` anymore.

This is called move, Rust will invalidate the first variable after you do the assignment.
Three tables: tables s1 and s2 representing those strings on the stack, respectively, and both pointi
This is wrong!

Clone

However, if you want to clone the data allocated on the heap you can call the `clone` method.

```
let s1 = String::from("hello");
let s2 = s1.clone();
```

Now `s2` will also contain a copy of the heap data pointed by `s1`.

What about variable on the stack?

If you use say primitives in Rust like an integer like so

```
let x = 5;
let y = x;
```

Both `x, y` are still valid after running line number 2, why? This is because primitive's data size like integer is known at compile time and are stored entirely on the stack, copying the actual value are quick and easy to make, so there is no reason to invalidate `x` afterward.

Function and ownership

If you decide to pass a variable data that's allocated on the heap to a function let's say, it will also carry out move, which means the variable that you pass into the function will no longer be valid after you do the function call like so:

```
fn main() {
let s = String::from("hello");
```

```
    take(s); // s is no longer validate after

    let x = 5;

    cant_take(x); // x is still valid because it is a copy, not a move
}

fn take(input: String) {
    // do something
}

fn cant_take(num: i32) {
    // do something
}
```

References and borrowing

So to solve what we have just talked about, instead of moving the variable into the function, we will give the function a reference to the variable, so that after the function call the variable with data allocated on the heap is still valid afterward.

> References uses the ampersand syntax, note this is not getting back the pointer, although, references are implemented via pointer, this is different than saying &num in C, which get you the pointer that points to the number.

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate(&s1);

    println("{}'s length is {}", s1, len);
}

fn calculate(s: &String) -> usize {
    s.len()
}
```

In this case, in order to pass a reference of a variable, we have to change the parameter type of the function from `String` to `&String` to denote that the parameter is indeed a reference to a String. This is called borrowing, it is borrowing the reference without moving it.

Now `s` will be a reference, which points to the same data that `s1` points to.

Mutable references

If you want to say append any data to the reference that is passed to a function, you have to add the mutable modifier otherwise, the reference would not be able to make any changes to the data on the heap.

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world"); // This will not work!
}
```

After adding `mut` modifier to both of the variable and the function header to signal that this function will change the String then it will work properly:

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

When you are doing mutable borrowing you can only do so when there is no other references to that same value:

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s; // second mutable borrow occurred, doesn't work
```

In this case `r2` is a second mutable borrower, but there is already a mutable borrower that existed already, hence the code will not compile.

In addition, if you are creating mutable references you cannot fix it with immutable references:

```
let mut s = String::from("hello");


let r1 = &s; // no problem
let r2 = &s; // a second immutable reference is fine
let r3 = &mut s; // error mixing immutable and mutable references
```

However, if you finish using a reference, (the scope of a reference starts from where it was introduced and continue through the last time that reference is used), then you can introduce say a mutable reference.

```
let mut s = String::from("hello");


let r1 = &s; // fine
let r2 = &s; // both immutable reference so is okay


println!("{} {}", r1, r2); // using the two immutable references, so their scope ends here


let r3 = &mut s; // introducing a mutable reference, it is okay because immutable references
ended the line before
println("{}", r3);
```

Dangling reference

If you are creating a local variable in a function, and then you attempt to return a reference to that variable from that function, Rust will prevent you from doing that because the data will be invalidated after the function is finished.

To resolve that problem, instead of returning a reference, return that variable directly, this will result in a move which won't deallocate that local variable if it is a move. However, if you return a reference, the local variable will be deallocated and that reference will become invalid. (In C you can return a pointer to a local variable after the function is finished, but dereferencing it will resulting in undefined behavior).

Revision #4
Created 2023-01-23 23:25:05 UTC by Tamarine
Updated 2023-01-28 17:47:14 UTC by Tamarine